Part-4

Docker Network & Docker Volumes

Docker has transformed the way developers build, ship, and run applications. Two of its key components—**Docker networks** and **Docker volumes**—play a crucial role in enabling seamless communication between containers and ensuring persistent storage for container data.

In this document, we'll dive deep into **Docker networks** and **Docker volumes**, exploring their concepts, types, and practical use cases with detailed examples. By the end of this, you'll have a comprehensive understanding of how to use both effectively in your Docker environment.

Part 1: Docker Networks

1. What is a Docker Network?

A **Docker network** is a communication bridge that connects Docker containers to each other and external systems. By default, containers are isolated from one another, but networks allow containers to communicate securely and efficiently.

When you launch a Docker container, it can be connected to one or more networks. Docker provides several types of networks that support different communication and isolation needs.

Why Docker Networks are Important

- **Container communication**: Docker networks facilitate communication between containers running on the same host or across different hosts.
- **Isolation**: Docker networks can isolate containers from each other based on specific networking needs.
- **Customization**: You can configure Docker networks with custom DNS, routing rules, and more.

---

2. Types of Docker Networks

Docker offers different types of networks, each designed for specific use cases:

2.1. Bridge Network (Default)

The **bridge** network is the default network type Docker uses when you start a container without specifying a network. It is a private internal network that containers connect to. Containers in a bridge network can communicate with each other, but they are isolated from containers on different networks.

Characteristics of Bridge Network:

- Default network when no other network is specified.
- Containers connected to the bridge network can communicate using IP addresses or container names.
- Containers cannot be accessed directly from the host machine without port mapping.

Example: Creating and Using a Bridge Network

1. **Create a custom bridge network**:

```
docker network create my-bridge-network
```

2. **Run a container and connect it to the custom network**:

```
docker run -d --name web-app --network my-bridge-network nginx
```

3. **Inspect the network to see connected containers**:

```
docker network inspect my-bridge-network
```

4. **Run another container and check connectivity**:

```
docker run -d --name db --network my-bridge-network mysql
```

You can now ping the db container from the web-app container by name, as Docker provides DNS resolution within the same network.

2.2. Host Network

In the **host** network mode, the container shares the host machine's networking stack. This means that the container's network is effectively the host's network. It's useful for cases where you need to maximize performance or need low-latency access to network interfaces.

Characteristics of Host Network:

- The container uses the host machine's network interfaces and ports.
- No isolation between the host and the container (security risk).
- Improved performance compared to bridge networks (no NAT overhead).

Example: Running a Container in Host Network

```
docker run -d --name nginx --network host nginx
```

In this case, the nginx container will run on the host's network interface and will be accessible at the same IP and port as the host system.

2.3. Overlay Network

The **overlay** network allows containers to communicate across different Docker hosts, enabling multi-host container communication. It is commonly used in **Docker Swarm** or **Kubernetes** environments for distributed applications.

Characteristics of Overlay Network:

- Enables multi-host networking for distributed systems.
- Containers on different hosts can communicate as if they were on the same network.
- Useful in orchestrated environments like Docker Swarm

- **Example: Creating and Using an Overlay Network**

First, ensure Docker Swarm is initialized:

```
docker swarm init
```

Then, create an overlay network:

```
docker network create -d overlay my-overlay-network
```

Finally, deploy services to this network:

```
docker service create --name web-app --network my-overlay-network nginx
```

2.4. Macvlan Network

The **macvlan** network allows you to assign a MAC address to containers, making them appear as physical devices on the network. This is useful when you want containers to behave like real devices on the network, each with its own IP address.

Characteristics of Macvlan Network:

- Each container gets its own MAC address and behaves like a physical device on the network.
- Containers are directly accessible from the external network.
- Requires more configuration compared to other network types.

Example: Creating a Macvlan Network

```
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 my-macvlan-network
```

This command creates a macvlan network on the eth0 interface of the host with a specific IP range. Containers connected to this network will behave like separate devices.

2.5. None Network

The **none** network disables all networking for the container. This is useful for highly isolated containers that do not require network access.

Characteristics of None Network:

- No network interfaces are provided to the container.
- Useful for security or specific use cases where isolation is required.

Example: Running a Container with No Network

```
docker run -d --name isolated-app --network none busybox
```

This container will run in complete isolation without any network access.

---

3. Docker Network Commands

Here are some useful Docker network commands:

List All Docker Networks:

```
docker network ls
```

Inspect a Docker Network:

```
docker network inspect <network_name>
```

Connect a Running Container to a Network:

```
docker network connect <network_name> <container_name>
```

Disconnect a Running Container from a Network:

```
docker network disconnect <network_name> <container_name>
```

---

4. Practical Use Case: Multi-Container Application

Let’s create a real-world scenario where multiple containers communicate using a custom bridge network.

Scenario:

You are building a simple web application with a front-end (Nginx) and a back-end database (MySQL). The two services need to communicate securely.

Steps:

1. **Create a Custom Bridge Network**:

```
docker network create webapp-network
```

2. **Run MySQL Container on the Network**:

```
docker run -d --name db \
--network webapp-network \
-e MYSQL_ROOT_PASSWORD=rootpass \
-e MYSQL_DATABASE=webapp \
mysql:latest
```

3. **Run Nginx Container on the Network**:

```
docker run -d --name webapp \
--network webapp-network \
-p 8080:80 nginx
```

4. **Verify Network Connectivity**: Access the running nginx container and ping the db container:

```
docker exec -it webapp ping db
```

By using Docker networks, we've allowed the web application to securely communicate with the database over a private network without exposing the database to the outside world.

Part 2: Docker Volumes

1. What is a Docker Volume?

Docker volumes are a way of persisting data generated and used by containers. When a container is deleted, its filesystem is also removed. However, volumes provide a way to persist data beyond the lifecycle of a container.

Docker volumes are stored on the host filesystem and can be mounted to one or more containers.

Why Use Volumes?

- **Persistence**: Volumes store data that needs to persist beyond the lifecycle of the container.
- **Shared Storage**: Volumes can be shared between multiple containers.
- **Isolation from Container Layers**: Data in volumes isn't affected by the container's image or the way containers are started.

---

2. Types of Docker Volumes

Docker provides several ways to persist data for containers:

2.1. Named Volumes

A **named volume** is created and managed by Docker. Named volumes are stored in Docker's managed location on the host system.

Example: Creating and Using Named Volumes

1. **Run a container with a named volume**:

```
docker run -d --name db \
-e MYSQL_ROOT_PASSWORD=rootpass \
-v my-db-volume:/var/lib/mysql \
mysql:latest
```

2. **Inspect the volume**:

```
docker volume inspect my-db-volume
```

Named volumes are stored in Docker’s managed location, typically /var/lib/docker/volumes/.

2.2. Bind Mounts

A **bind mount** maps a specific directory on the host system to a directory inside the container. This allows the container to directly access files and directories on the host.

Example: Using Bind Mounts

```
docker run -d --name webserver \
-v /home/user/site-content:/usr/share/nginx/html \
-p 8080:80 nginx
```

This mounts the /home/user/site-content directory from the host to /usr/share/nginx/html in the container, allowing the container to serve static content from the host filesystem.

2.3. tmpfs Mounts

A **tmpfs mount** stores data in memory rather than on disk. This is useful for sensitive data that you don’t want written to disk.

Example: Using tmpfs Mount

```
docker run -d --name tmpfs-container \
--tmpfs /app:rw,size=64m \
nginx
```

This creates a tmpfs mount at /app inside the container with a size limit of 64MB.

2.4. Anonymous Volumes

Anonymous volumes are volumes that Docker creates automatically when you run a container without explicitly specifying a volume or bind mount. These volumes have no explicit name and are given a randomly generated identifier by Docker. Anonymous volumes are useful when you want to ensure that your data is persisted but don't need to reference it directly outside of the container.

While they function similarly to named volumes, the key difference is that you cannot easily reuse or manage anonymous volumes since they are not explicitly named.

Characteristics of Anonymous Volumes:

- Automatically created when you specify a volume without a name.
- Managed by Docker, and Docker assigns a random name.
- Useful for temporary data or cases where you want Docker to handle volume management.
- They are automatically deleted when the container is removed with the --rm option, but they persist otherwise.

Example: Using Anonymous Volumes

If you run a container with an anonymous volume:

```
docker run -d --name nginx-container -v /usr/share/nginx/html nginx
```

In this example:

- Docker creates an anonymous volume for /usr/share/nginx/html inside the container.
- The volume is automatically created by Docker without a specified name, and the data will persist even if the container is stopped.

Inspecting Anonymous Volumes:

You can check the volumes created by Docker using:

```
docker volume ls
```

Anonymous volumes will appear with a random ID in the output, making it hard to track their use:

```
DRIVER    VOLUME NAME

local     02e1f7df459dbf5f7032d37f24d6453744fe

local     26a3f7df223dcf5640321f65db548a3623de
```

Anonymous volumes are especially useful for applications that need to store temporary data but don’t require direct management of the volume by the user.

Comparison of Docker Volume Types

Each type of Docker volume serves different use cases. Here’s a detailed comparison to help you understand when to use each type:

Volume Type	**Use Case**	**Pros**	**Cons**
Named Volumes	- Persistent storage for databases, configuration files, or critical data. - Easy to back up and move between hosts.	- Managed by Docker, stored in a default location. - Portable across different containers and hosts. - Easy to reference by name.	- Less direct control over the exact location of the data on the host system. - Must be manually cleaned up when not needed.
Anonymous Volumes	- Temporary storage or when you don't need to access the volume outside the container. - Short-term storage of runtime data.	- Automatically managed by Docker. - No need for manual naming or management. - Ideal for temporary data in containers.	- Difficult to reference outside of the container. - Automatically created and harder to track or manage directly. - Persists on disk unless cleaned up manually.

Volume Type	Use Case	Pros	Cons
Bind Mounts	- Development environments where you need access to host files (e.g., code, logs). - Testing or debugging locally with real-time file changes.	- Full control over where the data is stored on the host system. - Easy to work with during local development. - Provides direct access to host files.	- Less isolation from the host filesystem, which can lead to security risks. - Not portable across different hosts or environments. - More difficult to back up compared to named volumes.
tmpfs Mounts	- Temporary storage that only needs to exist in memory (e.g., sensitive data, caches).	- Fast storage as it is in-memory. - No data is written to disk, so it is more secure for sensitive data.	- Data is lost when the container stops or restarts. - Limited by system memory size.

When to Use Each Volume Type:

- **Named Volumes**:
  - Use when you need persistent, managed data storage that can easily be backed up or shared across multiple containers or hosts.
  - Ideal for production databases, configuration files, or any critical data that should persist across container lifecycles.
- **Anonymous Volumes**:
  - Use when you need temporary storage and don't need to refer to the data outside of the container.
  - Useful for quick deployments where you don’t want to manage volumes manually.
- **Bind Mounts**:
  - Use when you need real-time access to files on the host system, such as during development, debugging, or testing.
  - Ideal for development environments where you need to share code, logs, or configuration files between the host and container.
- **tmpfs Mounts**:
  - Use when you need temporary storage that is fast and doesn't need to be written to disk.
  - Ideal for sensitive data that should not be persisted on disk or cached data that can be regenerated.

3. Docker Volume Commands

Here are some common Docker volume commands:

List All Volumes:

```
docker volume ls
```

Create a Volume:

```
docker volume create my-volume
```

Inspect a Volume:

```
docker volume inspect my-volume
```

Remove a Volume:

```
docker volume rm my-volume
```

Remove Unused Volumes:

```
docker volume prune
```

---

4. Practical Use Case: Using Volumes in a Multi-Container Application

Let’s take the same web application scenario but add data persistence using Docker volumes.

Steps:

1. **Create a Named Volume for MySQL Data**:

```
docker volume create db-data
```

2. **Run MySQL Container with the Named Volume**:

```
docker run -d --name db \
--network webapp-network \
-e MYSQL_ROOT_PASSWORD=rootpass \
-e MYSQL_DATABASE=webapp \
-v db-data:/var/lib/mysql \
mysql:latest
```

3. **Run Nginx Container**:

```
docker run -d --name webapp \

--network webapp-network \

-p 8080:80 nginx
```

4. **Verify Data Persistence**: Even if the db container is stopped or removed, the MySQL data will persist in the db-data volume. You can restart the container, and the data will still be there:

```
docker stop db
docker rm db
docker run -d --name db \
--network webapp-network \
-e MYSQL_ROOT_PASSWORD=rootpass \
-e MYSQL_DATABASE=webapp \
-v db-data:/var/lib/mysql \
mysql:latest
```